Hadoop Material

Data!

We live in the data age. It's not easy to measure the total volume of data stored electronically, but an IDC estimate put the size of the "digital universe" at 0.18 zettabytes in 2006, and is forecasting a tenfold growth by 2011 to 1.8 zettabytes.

A zettabyte is 1021 bytes, or equivalently one thousand exabytes, one million petabytes, or one billion terabytes. That's roughly the same order of magnitude as one disk drive for every person in the world.

This flood of data is coming from many sources. Consider the following:

• The New York Stock Exchange generates about one terabyte of new trade data per day.

• Facebook hosts approximately 10 billion photos, taking up one petabyte of storage.

• Ancestry.com, the genealogy site, stores around 2.5 petabytes of data.

• The Internet Archive stores around 2 petabytes of data, and is growing at a rate of 20 terabytes per month.

Data Storage and Analysis

The problem is simple: while the storage capacities of hard drives have increased massively over the years, access speeds—the rate at which data can be read from drives— have not kept up.

- One typical drive from 1990 could store 1,370 MB of data and had a transfer speed of 4.4 MB/s, so you could read all the data from a full drive in around five minutes. Over 20 years later, one terabyte drives are the norm, but the transfer speed is around 100 MB/s, so it takes more than two and a half hours to read all the data off the disk. This is a long time to read all data on a single drive—and writing is even slower.
- The obvious way to reduce the time is to read from multiple disks at once. Imagine if we had 100 drives, each holding one hundredth of the data. Working in parallel, we could read the data in less than two minutes. Only using one hundredth of a disk may seem wasteful. But we can store one hundred datasets, each of which is one terabyte, and provide shared access to them.

There's more to being able to read and write data in parallel to or from multiple disks, though.

- **The first problem to solve is hardware failure:** as soon as you start using many pieces of hardware, the chance that one will fail is fairly high.

  A common way of avoiding data loss is through replication: redundant copies of the data are kept by the system so that in the event of failure, there is another copy available.
- The second problem is that most analysis tasks need to be able to combine the data in some way; data read from one disk may need to be combined with the data from any of the other 99 disks. Various distributed systems allow data to be combined from multiple sources, but doing this correctly is notoriously challenging. MapReduce provides a programming model that abstracts the problem from disk reads and writes, transforming it into a computation over sets of keys and values.

What is Hadoop?

Apache Hadoop is a new way for enterprise to store and analyze data.

Hadoop is an open-source project administered by the Apache Software Foundation. Hadoop's contributors work for some of the world's biggest technology companies. That diverse, motivated community has produced a genuinely innovative platform for consolidating, combining and understanding large-scale data in order to better comprehend the data deluge.

Enterprises today collect and generate more data than ever before. Relational and data warehouse products excel at OLAP and OLTP workloads over structured data. **Hadoop is designed to solve a different problem:** The fast, reliable analysis of both structured data and complex data. As a result, many enterprises deploy Hadoop alongside their legacy IT systems, which allows them to combine old data and new data sets in powerful new ways.

Technically, Hadoop consists of two key services: Reliable data storage using the Hadoop Distributed File System (HDFS) and high-performance parallel data processing using a technique called MapReduce. Hadoop runs on a collection of commodity, shared-nothing servers. You can add or remove servers in a Hadoop cluster at will; the system detects and compensates for hardware or system problems on any server. Hadoop, in other words, is self-healing. It can deliver data and can run large-scale, high-performance processing jobs in spite of system changes or failures.

A Brief History of Hadoop

Hadoop was created by Doug Cutting, the creator of Apache Lucene, the widely used text search library. Hadoop has its origins in Apache Nutch, an open source web search engine, itself a part of the Lucene project.

The Origin of the Name “Hadoop”

The name Hadoop is not an acronym; it’s a made-up name. The project’s creator, Doug Cutting, explains how the name came about:

The name my kid gave a stuffed yellow elephant. Short, relatively easy to spell and
Pronounce, meaningless, and not used elsewhere: those are my naming criteria.

Nutch was started in 2002, and a working crawler and search system quickly emerged. However, they realized that their architecture wouldn’t scale to the billions of pages on the Web.

- In 2003 that described the architecture of Google’s distributed filesystem, called GFS, which was being used in production at Google. GFS, or something like it, would solve their storage needs for the very large files generated as a part of the web crawl and indexing process. In particular, GFS would free up time being spent on administrative

tasks such as managing storage nodes. In 2004, they set about writing an open source implementation, the Nutch Distributed Filesystem (NDFS).

- In 2004, Google published the paper that introduced MapReduce to the world. Early in 2005, the Nutch developers had a working MapReduce implementation in Nutch, and by the middle of that year all the major Nutch algorithms had been ported to run using MapReduce and NDFS.

  NDFS and the MapReduce implementation in Nutch were applicable beyond the realm of search, and
- In February 2006 they moved out of Nutch to form an independent subproject of Lucene called Hadoop.
- This was demonstrated in February 2008 when Yahoo! announced that its production search index was being generated by a 10,000-core Hadoop cluster. In January 2008, Hadoop was made its own top-level project at Apache, confirming its success and its diverse, active community. By this time, Hadoop was being used by many other companies besides Yahoo!, such as Last.fm, Facebook, and the New York Times.
- In April 2008, Hadoop broke a world record to become the fastest system to sort a terabyte of data. Running on a 910-node cluster, Hadoop sorted one terabyte in 209 seconds (just under 3½ minutes), beating the previous year's winner of 297 seconds. In November of the same year, Google reported that its MapReduce implementation sorted one terabyte in 68 seconds.
- 2004—Initial versions of what is now Hadoop Distributed Filesystem and Map-Reduce implemented by Doug Cutting and Mike Cafarella.
- December 2005—Nutch ported to the new framework. Hadoop runs reliably on 20 nodes.
- January 2006—Doug Cutting joins Yahoo!.
- February 2006—Apache Hadoop project officially started to support the standalone development of MapReduce and HDFS.

Hadoop Core Components

1. HDFS

2. MapReduce

Hadoop Fundamentals

- **Moving computation to data, not the other way around**

Hadoop does this by sending the computation artifacts, typically JARs, to the data nodes when a task is being run. Hadoop also looks closely at where the data blocks are located, and tries to schedule the computation as close to the data is possible (Hadoop is rack-aware).

- **"Moving Computation is Cheaper than Moving Data"**

A computation requested by an application is much more efficient if it is executed near the data it operates on. This is especially true when the size of the data set is huge. This minimizes network congestion and increases the overall throughput of the system. The assumption is that it is often better to migrate the computation closer to where the data is located rather than moving the data to where the application is running. HDFS provides interfaces for applications to move themselves closer to where the data is located.
This fact in computer evolution is why moving computation to data is better than the other way around, especially when you're talking Big Data. The traditional approach of expensive and capable servers with SAN/NAS storage eventually hits the networking wall. A computer today is extraordinarily good at performing computations and storing large amounts of data. Contrast that with the fact that a cluster of such machines is extraordinarily bad at moving data around, and we have our answer to why moving computation to data is much better.

And "that's one of the fundamental reasons why Hadoop is a game-changing technology"

HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

Filesystems that manage the storage across a network of machines are called distributed filesystems. Hadoop comes with a distributed filesystem called HDFS.
The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware.

- It has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant.
- HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware.
- HDFS provides high throughput access to application data and is suitable for applications that have large data sets.

1. Assumptions and Goals:

- **Hardware Failure**

Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a non-trivial probability of failure means that some component of HDFS is always non-functional.

- **Streaming Data Access**

Applications that run on HDFS need streaming access to their data sets. They are not general purpose applications that typically run on general purpose file systems. HDFS is designed more for batch processing rather than interactive use by users. The emphasis is on high throughput of data access rather than low latency of data access.

- **Large Data Sets**

Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size. Thus, HDFS is tuned to support large files. It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster. It should support tens of millions of files in a single instance.

- **Simple Coherency Model**

HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. This assumption simplifies data coherency issues and enables high throughput data access. A Map/Reduce application or a web crawler application fits perfectly with this model. There is a plan to support appending-writes to files in the future.

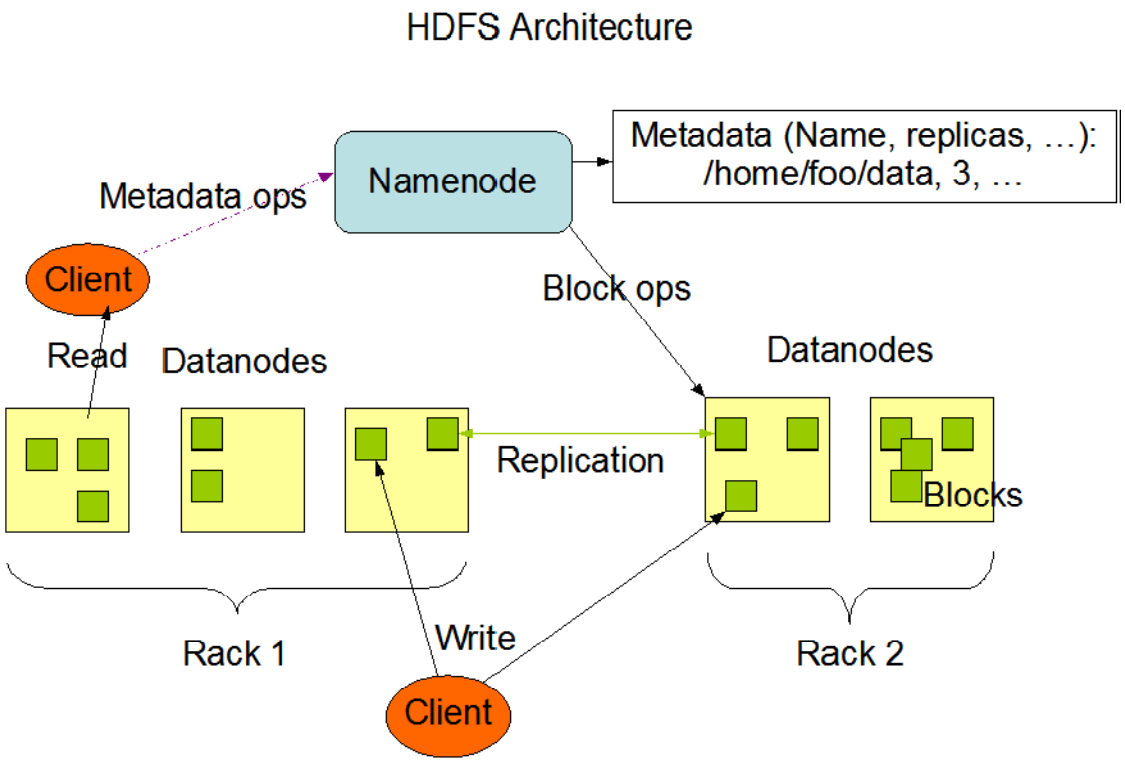
- **Portability Across Heterogeneous Hardware and Software Platforms**

HDFS has been designed to be easily portable from one platform to another. This facilitates widespread adoption of HDFS as a platform of choice for a large set of applications.

2. NameNode and DataNodes

3.

HDFS has master/slave architecture. An HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients. In addition, there are a number of DataNodes, usually one per node in the cluster, which manage storage attached to the nodes that they run on. HDFS exposes a file system namespace and allows user data to be stored in files. Internally, a file is split into one or more blocks and these blocks are stored in a set of DataNodes. The NameNode executes file system namespace operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the file system's clients. The DataNodes also perform block creation, deletion, and replication upon instruction from the NameNode.



The NameNode and DataNode are pieces of software designed to run on commodity machines. These machines typically run a GNU/Linux operating system (OS).

HDFS is built using the Java language; any machine that supports Java can run the NameNode or the DataNode software. Usage of the highly portable Java language means that HDFS can be deployed on a wide range of machines.

A typical deployment has a dedicated machine that runs only the NameNode software. Each of the other machines in the cluster runs one instance of the DataNode software. The architecture does not preclude running multiple DataNodes on the same machine but in a real deployment that is rarely the case.

The existence of a single NameNode in a cluster greatly simplifies the architecture of the system. The NameNode is the arbitrator and repository for all HDFS metadata. The system is designed in such a way that user data never flows through the NameNode.

4. The File System Namespace

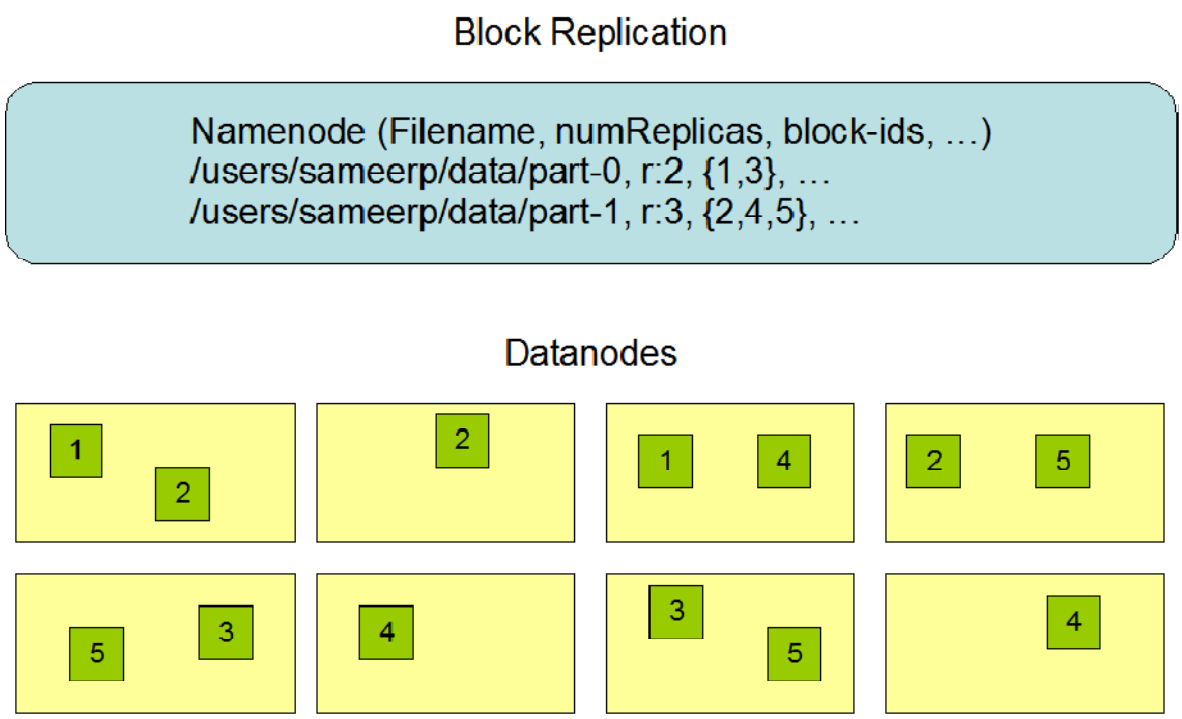
HDFS supports a traditional hierarchical file organization. A user or an application can create directories and store files inside these directories. The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file. HDFS does not yet implement user quotas. HDFS does not support hard links or soft links. However, the HDFS architecture does not preclude implementing these features.

The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the NameNode.

4. Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

The NameNode makes all decisions regarding replication of blocks. It periodically receives a Heartbeat and a Blockreport from each of the DataNodes in the cluster. Receipt of a Heartbeat implies that the DataNode is functioning properly. A Blockreport contains a list of all blocks on a DataNode.



- **Replica Placement:**

The placement of replicas is critical to HDFS reliability and performance. Optimizing replica placement distinguishes HDFS from most other distributed file systems. This is a feature that needs lots of tuning and experience. The purpose of a rack-aware replica placement policy is to improve data reliability, availability, and network bandwidth utilization.

Large HDFS instances run on a cluster of computers that commonly spread across many racks. Communication between two nodes in different racks has to go through switches. In most cases, network bandwidth between machines in the same rack is greater than network bandwidth between machines in different racks.

The NameNode determines the rack id each DataNode belongs to via the process outlined in Hadoop Rack Awareness. A simple but non-optimal policy is to place replicas on unique racks. This prevents losing data when an entire rack fails and allows use of bandwidth from multiple racks when reading data.

- **Safe mode**

On startup, the NameNode enters a special state called Safemode. Replication of data blocks does not occur when the NameNode is in the Safemode state. The NameNode receives Heartbeat and Blockreport messages from the DataNodes. A Blockreport contains the list of data blocks that a DataNode is hosting. Each block has a specified minimum number of replicas. A block is considered safely replicated when the minimum number of replicas of that data block has checked in with the NameNode. After a configurable percentage of safely replicated data blocks checks in with the NameNode (plus an additional 30 seconds), the NameNode exits the Safemode state. It then determines the list of data blocks (if any) that still have fewer than the specified number of replicas.

5. The Persistence of File System Metadata

The HDFS namespace is stored by the NameNode. The NameNode uses a transaction log called the EditLog to persistently record every change that occurs to file system metadata. For example, creating a new file in HDFS causes the NameNode to insert a record into the EditLog indicating this. Similarly, changing the replication factor of a file causes a new record to be inserted into the EditLog. The NameNode uses a file in its local host OS file system to store the EditLog. The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage. The FsImage is stored as a file in the NameNode's local file system too.

The NameNode keeps an image of the entire file system namespace and file Blockmap in memory. This key metadata item is designed to be compact, such that a NameNode with 4 GB of RAM is plenty to support a huge number of files and directories. When the NameNode starts up, it reads the FsImage and EditLog from disk, applies all the transactions from the EditLog to the in-memory representation of the FsImage, and flushes out this new version into a new FsImage on disk. It can then truncate the old EditLog because its transactions have been applied to the persistent FsImage. This process is called a checkpoint.

The DataNode stores HDFS data in files in its local file system. The DataNode has no knowledge about HDFS files. It stores each block of HDFS data in a separate file in its local file system. The DataNode does not create all files in the same directory. Instead, it uses a heuristic to determine the optimal number of files per directory and creates subdirectories appropriately. It is not optimal to create all local files in the same directory because the local file system might not be able to efficiently support a huge number of files in a single

all HDFS data blocks that correspond to each of these local files and sends this report to the NameNode: this is the Blockreport.

6. The Communication Protocols

All HDFS communication protocols are layered on top of the TCP/IP protocol. A client establishes a connection to a configurable TCP port on the NameNode machine. It talks the ClientProtocol with the NameNode. The DataNodes talk to the NameNode using the DataNode Protocol. A Remote Procedure Call (RPC) abstraction wraps both the Client Protocol and the DataNode Protocol. By design, the NameNode never initiates any RPCs. Instead, it only responds to RPC requests issued by DataNodes or clients.

7. Robustness

The primary objective of HDFS is to store data reliably even in the presence of failures. The three common types of failures are NameNode failures, DataNode failures and network partitions.

- **Data Disk Failure, Heartbeats and Re-Replication**

Each DataNode sends a Heartbeat message to the NameNode periodically. A network partition can cause a subset of DataNodes to lose connectivity with the NameNode. The NameNode detects this condition by the absence of a Heartbeat message. The NameNode marks DataNodes without recent Heartbeats as dead and does not forward any new IO requests to them. Any data that was registered to a dead DataNode is not available to HDFS anymore. DataNode death may cause the replication factor of some blocks to fall below their specified value. The NameNode constantly tracks which blocks need to be replicated and initiates replication whenever necessary. The necessity for re-replication may arise due to many reasons: a DataNode may become unavailable, a replica may become corrupted, a hard disk on a DataNode may fail, or the replication factor of a file may be increased.

- **Cluster Rebalancing**

The HDFS architecture is compatible with data rebalancing schemes. A scheme might automatically move data from one DataNode to another if the free space on a DataNode falls below a certain threshold. In the event of a sudden high demand for a particular file, a scheme might dynamically create additional replicas and rebalance other data in the cluster.
These types of data rebalancing schemes are not yet implemented.

- **Data Integrity**

It is possible that a block of data fetched from a DataNode arrives corrupted. This corruption can occur because of faults in a storage device, network faults, or buggy software. The HDFS

client software implements checksum checking on the contents of HDFS files. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace. When a client retrieves file contents it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file. If not, then the client can opt to retrieve that block from another DataNode that has a replica of that block.

- **Metadata Disk Failure**

The FsImage and the EditLog are central data structures of HDFS. A corruption of these files can cause the HDFS instance to be non-functional. For this reason, the NameNode can be configured to support maintaining multiple copies of the FsImage and EditLog. Any update to either the FsImage or EditLog causes each of the FsImages and EditLogs to get updated synchronously. This synchronous updating of multiple copies of the FsImage and EditLog may degrade the rate of namespace transactions per second that a NameNode can support.

However, this degradation is acceptable because even though HDFS applications are very data intensive in nature, they are not metadata intensive. When a NameNode restarts, it selects the latest consistent FsImage and EditLog to use.

The NameNode machine is a single point of failure for an HDFS cluster. If the NameNode machine fails, manual intervention is necessary. Currently, automatic restart and failover of the NameNode software to another machine is not supported.

- **Snapshots**

Snapshots support storing a copy of data at a particular instant of time. One usage of the snapshot feature may be to roll back a corrupted HDFS instance to a previously known good point in time. HDFS does not currently support snapshots but will in a future release.

8. Data Organization

- **Data Blocks**

HDFS is designed to support very large files. Applications that are compatible with HDFS are those that deal with large data sets. These applications write their data only once but they read it one or more times and require these reads to be satisfied at streaming speeds. HDFS supports write-once-read-many semantics on files. A typical block size used by HDFS is 64 MB. Thus, an HDFS file is chopped up into 64 MB chunks, and if possible, each chunk will reside on a different DataNode.

- **Staging**

A client request to create a file does not reach the NameNode immediately. In fact, initially the HDFS client caches the file data into a temporary local file. Application writes are transparently redirected to this temporary local file. When the local file accumulates data worth over one HDFS block size, the client contacts the NameNode. The NameNode inserts the file name into the file system hierarchy and allocates a data block for it. The NameNode responds to the client request with the identity of the DataNode and the destination data block. Then the client flushes the block of data from the local temporary file to the specified DataNode. When a file is closed, the remaining un-flushed data in the temporary local file is transferred to the DataNode. The client then tells the NameNode that the file is closed. At this point, the NameNode commits the file creation operation into a persistent store. If the NameNode dies before the file is closed, the file is lost.

The above approach has been adopted after careful consideration of target applications that run on HDFS. These applications need streaming writes to files. If a client writes to a remote file directly without any client side buffering, the network speed and the congestion in the network impacts throughput considerably. This approach is not without precedent. Earlier distributed file systems, e.g. AFS, have used client side caching to improve performance. A POSIX requirement has been relaxed to achieve higher performance of data uploads.

9. Accessibility

HDFS can be accessed from applications in many different ways. Natively, HDFS provides a Java API for applications to use. A C language wrapper for this Java API is also available. In addition, an HTTP browser can also be used to browse the files of an HDFS instance. Work is in progress to expose HDFS through the WebDAV protocol.

- **FS Shell**

HDFS allows user data to be organized in the form of files and directories. It provides a command line interface called FS shell that lets a user interact with the data in HDFS.

- **Browser Interface**

A typical HDFS install configures a web server to expose the HDFS namespace through a configurable TCP port. This allows a user to navigate the HDFS namespace and view the contents of its files using a web browser.

10. Space Reclamation

10.1. File Deletes and Undeletes

When a file is deleted by a user or an application, it is not immediately removed from HDFS. Instead, HDFS first renames it to a file in the /trash directory. The file can be restored quickly as long as it remains in /trash. A file remains in /trash for a configurable amount of time. After the expiry of its life in /trash, the NameNode deletes the file from the HDFS

namespace. The deletion of a file causes the blocks associated with the file to be freed. Note that there could be an appreciable time delay between the time a file is deleted by a user and the time of the corresponding increase in free space in HDFS.
A user can Undelete a file after deleting it as long as it remains in the /trash directory. If a user wants to undelete a file that he/she has deleted, he/she can navigate the /trash directory and retrieve the file. The /trash directory contains only the latest copy of the file that was deleted. The /trash directory is just like any other directory with one special feature: HDFS applies specified policies to automatically delete files from this directory. The current default policy is to delete files from /trash that are more than 6 hours old. In the future, this policy will be configurable through a well-defined interface.

10.2. Decrease Replication Factor

When the replication factor of a file is reduced, the NameNode selects excess replicas that can be deleted. The next Heartbeat transfers this information to the DataNode. The DataNode then removes the corresponding blocks and the corresponding free space appears in the cluster. Once again, there might be a time delay between the completion of the setReplication API call and the appearance of free space in the cluster.

File I/O Operations and Replica Management

- **File Read and Write**

An application adds data to HDFS by creating a new file and writing the data to it. After the file is closed, the bytes written cannot be altered or removed except that new data can be added to the file by reopening the file for append. HDFS implements a single-writer, multiple-reader model.

An HDFS file consists of blocks. When there is a need for a new block, the NameNode allocates a block with a unique block ID and determines a list of DataNodes to host replicas of the block. The DataNodes form a pipeline, the order of which minimizes the total network distance from the client to the last DataNode. Bytes are pushed to the pipeline as a sequence of packets. The bytes that an application writes first buffer at the client side. After a packet buffer is filled (typically 64 MB), the data are pushed to the pipeline. The next packet can be pushed to the pipeline before receiving the acknowledgment for the previous packets. The number of outstanding packets is limited by the outstanding packets window size of the client.

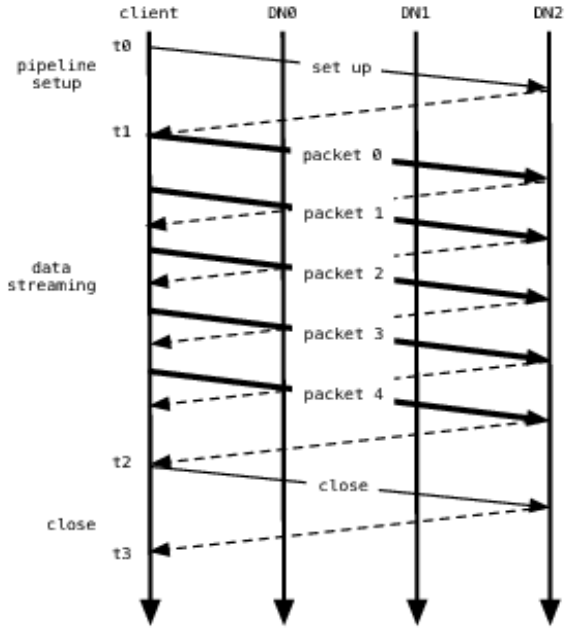


Figure: Data Pipeline While Writing a Block

If no error occurs, block construction goes through three stages as shown in Figure illustrating a pipeline of three DataNodes (DN) and a block of five packets. In the picture, bold lines represent data packets, dashed lines represent acknowledgment messages, and thin lines represent control messages to setup and close the pipeline. Vertical lines represent activity at the client and the three DataNodes where time proceeds from top to bottom. From t0 to t1 is the pipeline setup stage. The interval t1 to t2 is the data streaming stage, where t1 is the time when the first data packet gets sent and t2 is the time that the acknowledgment to the last packet gets received. Here an hflush operation transmits packet 2. The hflush indication travels with the packet data and is not a separate operation. The final interval t2 to t3 is the pipeline close stage for this block.

When a client opens a file to read, it fetches the list of blocks and the locations of each block replica from the NameNode. The locations of each block are ordered by their distance from the reader. When reading the content of a block, the client tries the closest replica first. If the read attempt fails, the client tries the next replica in sequence. A read may fail if the target DataNode is unavailable, the node no longer hosts a replica of the block, or the replica is found to be corrupt when checksums are tested.

The design of HDFS I/O is particularly optimized for batch processing systems, like MapReduce, which require high throughput for sequential reads and writes. Ongoing efforts will improve read/write response time for applications that require real-time data streaming or random access.

- **Block Placement**

For a large cluster, it may not be practical to connect all nodes in a flat topology. A common practice is to spread the nodes across multiple racks. Nodes of a rack share a switch, and rack switches are connected by one or more core switches. Communication between two nodes in different racks has to go through multiple switches. In most cases, network bandwidth between nodes in the same rack is greater than network bandwidth between nodes in different racks.

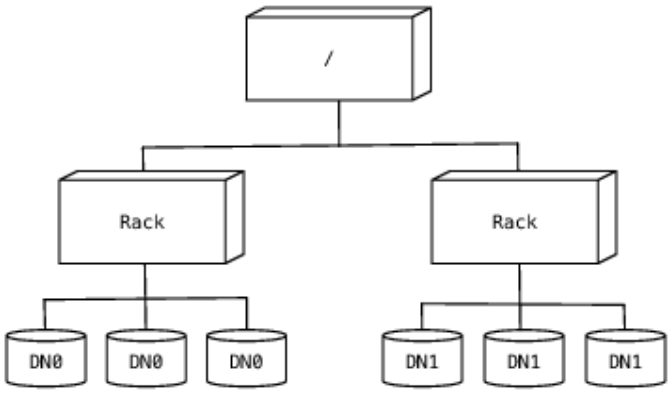


Fig: Cluster Topology

HDFS estimates the network bandwidth between two nodes by their distance. The distance from a node to its parent node is assumed to be one. A distance between two nodes can be calculated by summing the distances to their closest common ancestor. A shorter distance between two nodes means greater bandwidth they can use to transfer data.

HDFS allows an administrator to configure a script that returns a node's rack identification given a node's address. The NameNode is the central place that resolves the rack location of each DataNode. When a DataNode registers with the NameNode, the NameNode runs the configured script to decide which rack the node belongs to. If no such a script is configured, the NameNode assumes that all the nodes belong to a default single rack.

The placement of replicas is critical to HDFS data reliability and read/write performance. A good replica placement policy should improve data reliability, availability, and network bandwidth utilization. Currently HDFS provides a configurable block placement policy interface so that the users and researchers can experiment and test alternate policies that are optimal for their applications.

- **Replication Management**

The NameNode endeavors to ensure that each block always has the intended number of replicas. The NameNode detects that a block has become under- or over-replicated when a block report from a DataNode arrives. When a block becomes over replicated, the NameNode

chooses a replica to remove. The NameNode will prefer not to reduce the number of racks that host replicas, and secondly prefer to remove a replica from the DataNode with the least amount of available disk space. The goal is to balance storage utilization across DataNodes without reducing the block's availability.

When a block becomes under-replicated, it is put in the replication priority queue. A block with only one replica has the highest priority, while a block with a number of replicas that is greater than two thirds of its replication factor has the lowest priority. A background thread periodically scans the head of the replication queue to decide where to place new replicas. Block replication follows a similar policy as that of new block placement. If the number of existing replicas is one, HDFS places the next replica on a different rack. In case that the block has two existing replicas, if the two existing replicas are on the same rack, the third replica is placed on a different rack; otherwise, the third replica is placed on a different node in the same rack as an existing replica. Here the goal is to reduce the cost of creating new replicas.

The NameNode also makes sure that not all replicas of a block are located on one rack. If the NameNode detects that a block's replicas end up at one rack, the NameNode treats the block as mis-replicated and replicates the block to a different rack using the same block placement policy described above. After the NameNode receives the notification that the replica is created, the block becomes over-replicated. The NameNode then will decides to remove an old replica because the over-replication policy prefers not to reduce the number of racks.

- **Balancer**

HDFS block placement strategy does not take into account DataNode disk space utilization. This is to avoid placing new—more likely to be referenced—data at a small subset of the DataNodes with a lot of free storage. Therefore data might not always be placed uniformly across DataNodes. Imbalance also occurs when new nodes are added to the cluster.

The balancer is a tool that balances disk space usage on an HDFS cluster. It takes a threshold value as an input parameter, which is a fraction between 0 and 1. A cluster is balanced if, for each DataNode, the utilization of the node differs from the utilization of the whole cluster by no more than the threshold value.

The tool is deployed as an application program that can be run by the cluster administrator. It iteratively moves replicas from DataNodes with higher utilization to DataNodes with lower utilization. One key requirement for the balancer is to maintain data availability. When choosing a replica to move and deciding its destination, the balancer guarantees that the decision does not reduce either the number of replicas or the number of racks.

The balancer optimizes the balancing process by minimizing the inter-rack data copying. If the balancer decides that a replica A needs to be moved to a different rack and the destination

rack happens to have a replica B of the same block, the data will be copied from replica B instead of replica A.

A configuration parameter limits the bandwidth consumed by rebalancing operations. The higher the allowed bandwidth, the faster a cluster can reach the balanced state, but with greater competition with application processes.

- **Block Scanner**

Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data. In each scan period, the block scanner adjusts the read bandwidth in order to complete the verification in a configurable period. If a client reads a complete block and checksum verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica.

The verification time of each block is stored in a human-readable log file. At any time there are up to two files in the top-level DataNode directory, the current and previous logs. New verification times are appended to the current file. Correspondingly, each DataNode has an in-memory scanning list ordered by the replica's verification time.

Whenever a read client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt, but does not schedule deletion of the replica immediately. Instead, it starts to replicate a good copy of the block. Only when the good replica count reaches the replication factor of the block the corrupt replica is scheduled to be removed. This policy aims to preserve data as long as possible. So even if all replicas of a block are corrupt, the policy allows the user to retrieve its data from the corrupt replicas.

- **Decommissioning**

The cluster administrator specifies list of nodes to be decommissioned. Once a DataNode is marked for decommissioning, it will not be selected as the target of replica placement, but it will continue to serve read requests. The NameNode starts to schedule replication of its blocks to other DataNodes. Once the NameNode detects that all blocks on the decommissioning DataNode are replicated, the node enters the decommissioned state. Then it can be safely removed from the cluster without jeopardizing any data availability.

- **Inter-Cluster Data Copy**

When working with large datasets, copying data into and out of a HDFS cluster is daunting. HDFS provides a tool called DistCp for large inter/intra-cluster parallel copying. It is a MapReduce job; each of the map tasks copies a portion of the source data into the destination filesystem. The MapReduce framework automatically handles parallel task scheduling, error detection and recovery.

NAME NODE CONSIDERATIONS

➢ Marrying storage and compute

Over the past decade IT organizations have standardized on blades and SANs (Storage Area Networks) to satisfy their grid and processing-intensive workloads. While this model makes a lot of sense for a number of standard applications such as web servers, app servers, smaller structured databases and simple ETL (Extract, Transform, Load) the requirements for infrastructure has been changing as the amount of data and number of users has grown. Web servers now have caching tiers, databases have gone massively parallel with local disk, and ETL jobs are pushing more data than they can handle locally. Hardware vendors have created innovative systems to address these requirements including storage blades, SAS (Serial Attached SCSI) switches, external SATA arrays and larger capacity rack units.

Hadoop was designed based on a new approach to storing and processing complex data. Instead of relying on a SAN for massive storage and reliability then moving it to a collection of blades for processing, Hadoop handles large data volumes and reliability in the software tier. Hadoop distributes data across a cluster of balanced machines and uses replication to ensure data reliability and fault tolerance. Because data is distributed on machines with compute power, processing can be sent directly to the machines storing the data. Since each machine in a Hadoop cluster both stores and processes data, they need to be configured to satisfy both data storage and processing requirements.

➢ Why workloads matter

In nearly all cases, a MapReduce job will either encounter a bottleneck reading data from disk or from the network (known as an IO-bound job) or in processing data (CPU-bound). An example of an IO-bound job is sorting, which requires very little processing (simple comparisons) and a lot of reading and writing to disk. An example of a CPU-bound job is classification, where some input data is processed in very complex ways to determine anontology.

➢ How to pick hardware for your Hadoop cluster

The first step in choosing a machine configuration is to understand the type of hardware your operations team already manages. Operations teams often have opinions about new machine purchases and will prefer to work with hardware that they're already familiar with. Hadoop is not the only system that benefits from efficiencies of scale. Remember to plan on using balanced hardware for an initial cluster when new to Hadoop and if you do not yet understand your workload.

There are four types of nodes in a basic Hadoop cluster. We refer here to a node as a machine performing a particular task. Most of the machines will function as both datanodes and tasktrackers. As we described, these nodes both store data and perform processing functions. We recommend the following specifications for datanodes/tasktrackers in a balanced Hadoop cluster:

- 4 1TB hard disks in a JBOD (Just a Bunch Of Disks) configuration
- 2 quad core CPUs, running at least 2-2.5GHz
- 16-24GBs of RAM (24-32GBs if you're considering HBase)
- Gigabit Ethernet

The namenode is responsible for coordinating data storage on the cluster and the jobtracker for coordinating data processing. The last type of node is the secondarynamenode, which can be colocated on the namenode machine for small clusters, and will run on the same hardware as the namenode for larger clusters. We recommend our customers purchase hardened machines for running the namenodes and jobtrackers, with redundant power and enterprise-grade RAIDed disks. Namenodes also require more RAM relative to the number of data blocks in the cluster. A good rule of thumb is to assume 1GB of namenode memory for every one million blocks stored in the distributed file system. With 100 datanodes in a cluster, 32GBs of RAM on the namenode provides plenty of room to grow. We also recommend having a standby machine to replace the namenode or jobtracker, in the case when one of these fails suddenly.

When you expect your Hadoop cluster to grow beyond 20 machines we recommend that the initial cluster be configured as it were to span two racks, where each rack has a top of rack gigabit switch, and those switches are connected with a 10 GigE interconnect or core switch. Having two logical racks gives the operations team a better understand of the network requirements for inner-rack, and cross-rack communication.

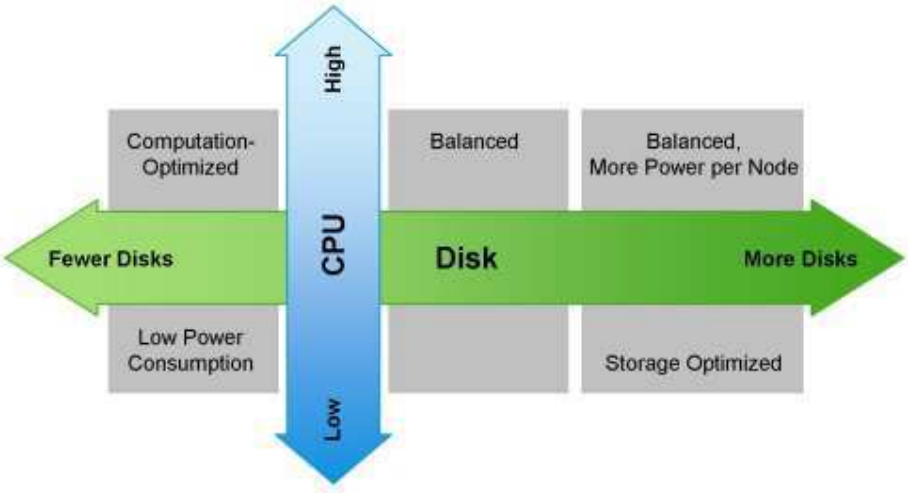
With a Hadoop cluster in place the team can start identifying workloads and prepare to benchmark those workloads to identify CPU and IO bottlenecks. After some time benchmarking and monitoring, the team will have a good understanding as to how additional machines should be configured. It is common to have heterogeneous Hadoop clusters especially as they grow in size. Starting with a set of machines that are not perfect for your workload will not be a waste.

Below is a list of various hardware configurations for different workloads, including our original "base" recommendation:

- Light Processing Configuration (1U/machine): Two quad core CPUs, 8GB memory, and 4 disk drives (1TB or 2TB). Note that CPU-intensive work such as natural language processing involves loading large models into RAM before processing data and should be configured with 2GB RAM/core instead of 1GB RAM/core.

- Balanced Compute Configuration (1U/machine): Two quad core CPUs, 16 to 24GB memory, and 4 disk drives (1TB or 2TB) directly attached using the motherboard controller. These are often available as twins with two motherboards and 8 drives in a single 2U cabinet.
- Storage Heavy Configuration (2U/machine): Two quad core CPUs, 16 to 24GB memory, and 12 disk drives (1TB or 2TB). The power consumption for this type of machine starts around ~200W in idle state and can go as high as ~350W when active.
- Compute Intensive Configuration (2U/machine): Two quad core CPUs, 48-72GB memory, and 8 disk drives (1TB or 2TB). These are often used when a combination of large in-memory models and heavy reference data caching is required.

Note that we expect to adopt 6 and 8 core configurations as they arrive. The following diagram shows how a machine should be configured according to workload:



➢ Other hardware considerations

When we encounter applications that produce large amounts of intermediate data–on the order of the same amount as is read in–we recommend two ports on a single Ethernet card or two channel-bonded Ethernet cards to provide 2 Gbps per machine. Alternatively for customers who have already moved to 10 Gigabit Ethernet or Infiniband, these solutions can be used to address network bound workloads. Be sure that your operating system and BIOS are compatible if you're considering switching to 10 Gigabit Ethernet.

When computing memory requirements, factor in that Java uses up to 10% for managing the virtual machine. We recommend configuring Hadoop to use strict heap size restrictions in order to avoid memory swapping to disk. Swapping greatly impacts MapReduce job performance and can be avoided by configuring machines with more RAM.

It is also important to optimize RAM for the memory channel width. For example, when using dual-channel memory each machine should be configured with pairs of DIMMs. With

triple-channel memory each machine should have triplets of DIMMs. This means a machine might end up with 18GBs (9x2GB) of RAM instead of 16GBs (4x4GB).

Persistent Data Structures

As an administrator, it is invaluable to have a basic understanding of how the components of HDFS—the namenode, the secondary namenode, and the datanodes— organize their persistent data on disk. Knowing which files are which can help you diagnose problems or spot that something is awry.

Namenode directory structure

A newly formatted namenode creates the following directory structure:

```
${dfs.name.dir}/
└── current/
├── VERSION
├── edits
├── fsimage
└── fstime
```

the dfs.name.dir property is a list of directories, with the same contents mirrored in each directory. This mechanism provides resilience, particularly if one of the directories is an NFS mount, as is recommended.

The VERSION file is a Java properties file that contains information about the version of HDFS that is running. Here are the contents of a typical file:

```
#Tue Mar 10 19:21:36 GMT 2009
namespaceID=134368441
cTime=0
storageType=NAME_NODE
layoutVersion=-18
```

The layoutVersion is a negative integer that defines the version of HDFS's persistent data structures. This version number has no relation to the release number of the Hadoop distribution. Whenever the layout changes, the version number is decremented (for example, the version after −18 is −19). When this happens, HDFS needs to be upgraded, since a newer namenode (or datanode) will not operate if its storage layout is an older version.

The namespaceID is a unique identifier for the filesystem, which is created when the filesystem is first formatted. The namenode uses it to identify new datanodes, since they will not know the namespaceID until they have registered with the namenode.

The cTime property marks the creation time of the namenode's storage. For newly formatted storage, the value is always zero, but it is updated to a timestamp whenever the filesystem is upgraded.

The storageType indicates that this storage directory contains data structures for a namenode. The other files in the namenode's storage directory are edits, fsimage, and fstime. These are all binary files that use Hadoop Writable objects as their serialization format. To

understand what these files are for, we need to dig into the workings of the namenode a little more.

The filesystem image and edit log

When a filesystem client performs a write operation (such as creating or moving a file), it is first recorded in the edit log. The namenode also has an in-memory representation of the filesystem metadata, which it updates after the edit log has been modified. The in-memory metadata is used to serve read requests.

The edit log is flushed and synced after every write before a success code is returned to the client. For namenodes that write to multiple directories, the write must be flushed and synced to every copy before returning successfully. This ensures that no operation is lost due to machine failure.

The fsimage file is a persistent checkpoint of the filesystem metadata. However, it is not updated for every filesystem write operation, because writing out the fsimage file, which can grow to be gigabytes in size, would be very slow. This does not compromise resilience, however, because if the namenode fails, then the latest state of its metadata can be reconstructed by loading the fsimage from disk into memory, and then applying each of the operations in the edit log. In fact, this is precisely what the namenode does when it starts up.

As described, the edits file would grow without bound. Though this state of affairs would have no impact on the system while the namenode is running, if the namenode were restarted, it would take a long time to apply each of the operations in its (very long) edit log. During this time, the filesystem would be offline, which is generally undesirable.

The solution is to run the secondary namenode, whose purpose is to produce checkpoints of the primary's in-memory filesystem metadata.

The checkpointing process proceeds as follows:

1. The secondary asks the primary to roll its edits file, so new edits go to a new file.
2. The secondary retrieves fsimage and edits from the primary (using HTTP GET).
3. The secondary loads fsimage into memory, applies each operation from edits, then creates a new consolidated fsimage file.
4. The secondary sends the new fsimage back to the primary (using HTTP POST).
5. The primary replaces the old fsimage with the new one from the secondary and the old edits file with the new one i started in step 1. It also updates the fstime file to record the time that the checkpoint was taken.

At the end of the process, the primary has an up-to-date fsimage file and a shorter edits file (it is not necessarily empty, as it may have received some edits while the checkpoint was being taken). It is possible for an administrator to run this process manually while the namenode is in safe mode, using the hadoop dfsadmin -saveNamespace command.

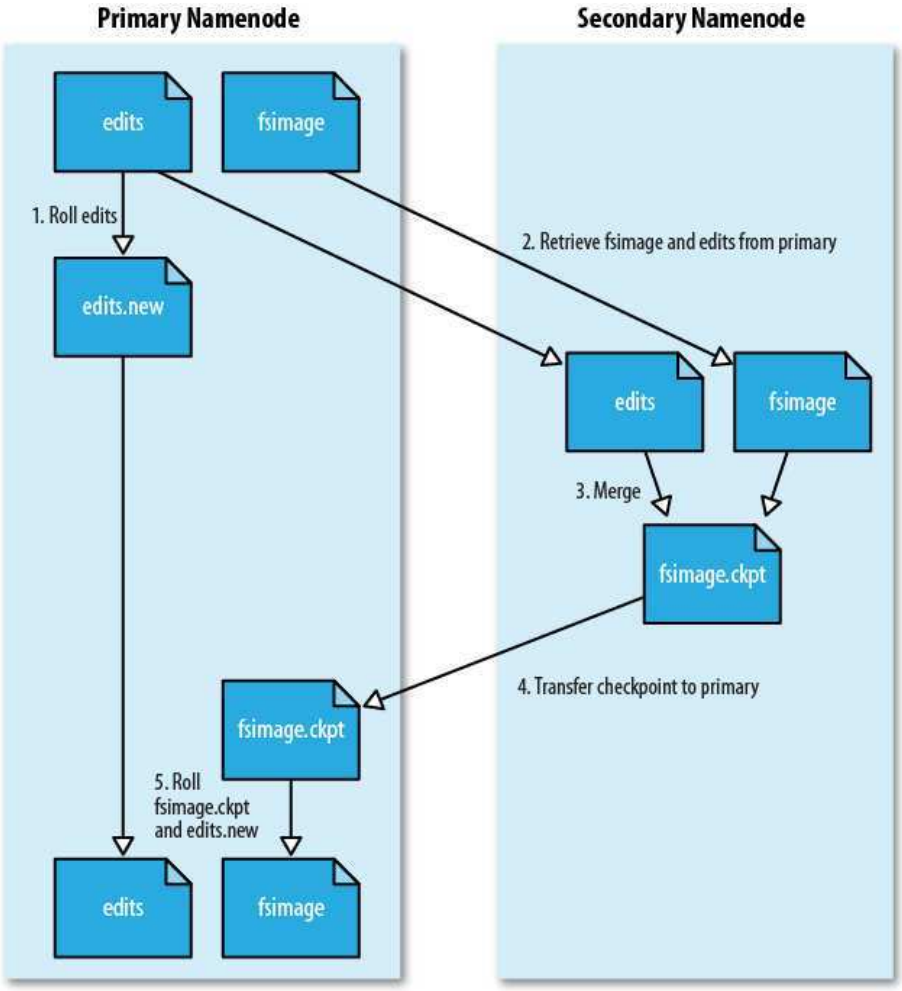


Fig: The check pointing process

This procedure makes it clear why the secondary has similar memory requirements to the primary (since it loads the fsimage into memory), which is the reason that the secondary needs a dedicated machine on large clusters.

The schedule for checkpointing is controlled by two configuration parameters. The secondary namenode checkpoints every hour (fs.checkpoint.period in seconds) or sooner if the edit log has reached 64 MB (fs.checkpoint.size in bytes), which it checks every five minutes.

Secondary namenode directory structure

A useful side effect of the checkpointing process is that the secondary has a checkpoint at the end of the process, which can be found in a subdirectory called previous.checkpoint.

This can be used as a source for making (stale) backups of the namenode's metadata:

${fs.checkpoint.dir}/

```
├── current/
│   ├── VERSION
│   ├── edits
│   ├── fsimage
│   └── fstime
└── previous.checkpoint/
├── VERSION
├── edits
├── fsimage
└── fstime
```

The layout of this directory and of the secondary's current directory is identical to the namenode's. This is by design, since in the event of total namenode failure (when there are no recoverable backups, even from NFS), it allows recovery from a secondary namenode. This can be achieved either by copying the relevant storage directory to a new namenode or, if the secondary is taking over as the new primary namenode, by using the -importCheckpoint option when starting the namenode daemon. The -importCheckpoint option will load the namenode metadata from the latest checkpoint in the directory defined by the fs.checkpoint.dir property, but only if there is no metadata in the dfs.name.dir directory, to ensure that there is no risk of overwriting precious metadata.

➢ Datanode directory structure

Unlike namenodes, datanodes do not need to be explicitly formatted, because they create their storage directories automatically on startup. Here are the key files and directories:

```
${dfs.data.dir}/
└── current/
├── VERSION
├── blk_<id_1>
├── blk_<id_1>.meta
├── blk_<id_2>
├── blk_<id_2>.meta
├── ...
├── blk_<id_64>
├── blk_<id_64>.meta
├── subdir0/
├── subdir1/
├── ...
└── subdir63/
```

A datanode's VERSION file is very similar to the namenode's:

```
#Tue Mar 10 21:32:31 GMT 2009
namespaceID=134368441
storageID=DS-547717739-172.16.85.1-50010-1236720751627
```

cTime=0
storageType=DATA_NODE
layoutVersion=-18

The namespaceID, cTime, and layoutVersion are all the same as the values in the namenode (in fact, the namespaceID is retrieved from the namenode when the datanode first connects). The storageID is unique to the datanode (it is the same across all storage directories) and is used by the namenode to uniquely identify the datanode. The storageType identifies this directory as a datanode storage directory.

The other files in the datanode's current storage directory are the files with the blk_ prefix. There are two types: the HDFS blocks themselves (which just consist of the file's raw bytes) and the metadata for a block (with a .meta suffix). A block file just consists of the raw bytes of a portion of the file being stored; the metadata file is made up of a header with version and type information, followed by a series of checksums for sections of the block.

When the number of blocks in a directory grows to a certain size, the datanode creates a new subdirectory in which to place new blocks and their accompanying metadata. It creates a new subdirectory every time the number of blocks in a directory reaches 64 (set by the dfs.datanode.numblocks configuration property). The effect is to have a tree with high fan-out, so even for systems with a very large number of blocks, the directories will be only a few levels deep. By taking this measure, the datanode ensures that there is a manageable number of files per directory, which avoids the problems that most operating systems encounter when there are a large number of files (tens or hundreds of thousands) in a single directory.

If the configuration property dfs.data.dir specifies multiple directories on differen drives, blocks are written to each in a round-robin fashion. Note that blocks are not replicated on each drive on a single datanode; instead, block replication is across distinct datanodes.

➢ Safe Mode

When the namenode starts, the first thing it does is load its image file (fsimage) into memory and apply the edits from the edit log (edits). Once it has reconstructed a consistent in-memory image of the filesystem metadata, it creates a new fsimage file (effectively doing the checkpoint itself, without recourse to the secondary namenode) and an empty edit log. Only at this point does the namenode start listening for RPC and HTTP requests. However, the namenode is running in safe mode, which means that it offers only a read-only view of the filesystem to clients.

Recall that the locations of blocks in the system are not persisted by the namenode; this information resides with the datanodes, in the form of a list of the blocks it is storing. During normal operation of the system, the namenode has a map of block locations stored in memory. Safe mode is needed to give the datanodes time to check in to the namenode with their block lists, so the namenode can be informed of enough block locations to run the filesystem effectively. If the namenode didn't wait for enough datanodes to check in, then it would start the process of replicating blocks to new datanodes, which would be unnecessary in most cases (because it only needed to wait for the extra datanodes to check in) and would put a great strain on the cluster's resources.

Indeed, while in safe mode, the namenode does not issue any block-replication or deletion instructions to datanodes.

Safe mode is exited when the minimal replication condition is reached, plus an extension time of 30 seconds. The minimal replication condition is when 99.9% of the blocks in the whole filesystem meet their minimum replication level (which defaults to one and is set by dfs.replication.min.

When you are starting a newly formatted HDFS cluster, the namenode does not go into safe mode, since there are no blocks in the system.

➢ Entering and leaving safe mode

To see whether the namenode is in safe mode, you can use the dfsadmin command:

$ **hadoop dfsadmin -safemode get**

Safe mode is ON

The front page of the HDFS web UI provides another indication of whether the namenode is in safe mode.

Sometimes you want to wait for the namenode to exit safe mode before carrying out a command, particularly in scripts. The wait option achieves this:

$ hadoop dfsadmin -safemode wait

command to read or write a file

An administrator has the ability to make the namenode enter or leave safe mode at any time. It is sometimes necessary to do this when carrying out maintenance on the cluster or after upgrading a cluster to confirm that data is still readable. To enter safe mode, use the following command:

$ **hadoop dfsadmin -safemode enter**

Safe mode is ON

You can use this command when the namenode is still in safe mode while starting up to ensure that it never leaves safe mode. Another way of making sure that the namenode stays in safe mode indefinitely is to set the property dfs.safemode.threshold.pct to a value over one.

You can make the namenode leave safe mode by using:

$ **hadoop dfsadmin -safemode leave**

Safe mode is OFF

➢ Audit Logging

HDFS can log all filesystem access requests, a feature that some organizations require for auditing purposes. Audit logging is implemented using log4j logging at the INFO level. In the default configuration it is disabled, as the log threshold is set to WARN in log4j.properties:

log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit=WARN You can enable audit logging by replacing WARN with INFO, and the result will be a log line written to the namenode's log for every HDFS event. Here's an example for a list status request on /user/tom:

It is a good idea to configure log4j so that the audit log is written to a separate file and isn't mixed up with the namenode's other log entries.

➢ Tools

dfsadmin

The dfsadmin tool is a multipurpose tool for finding information about the state of HDFS, as well as for performing administration operations on HDFS. It is invoked as hadoop dfsadmin and requires superuser privileges.

Filesystem check (fsck)

Hadoop provides an fsck utility for checking the health of files in HDFS. The tool looks for blocks that are missing from all datanodes, as well as under- or over-replicated blocks. Here is an example of checking the whole filesystem for a small cluster:

$ **hadoop fsck /**

.....Status: HEALTHY
Total size: 511799225 B
Total dirs: 10
Total files: 22
Total blocks (validated): 22 (avg. block size 23263601 B)
Minimally replicated blocks: 22 (100.0 %)
Over-replicated blocks: 0 (0.0 %)
Under-replicated blocks: 0 (0.0 %)
Mis-replicated blocks: 0 (0.0 %)
Default replication factor: 3
Average block replication: 3.0
Corrupt blocks: 0
Missing replicas: 0 (0.0 %)
Number of data-nodes: 4
Number of racks: 1
The filesystem under path '/' is HEALTHY

fsck recursively walks the filesystem namespace, starting at the given path (here the filesystem root), and checks the files it finds. It prints a dot for every file it checks. To check a file, fsck retrieves the metadata for the file's blocks and looks for problems or inconsistencies. Note that fsck retrieves all of its information from the namenode; it does not communicate with any datanodes to actually retrieve any block data.

Most of the output from fsck is self-explanatory, but here are some of the conditions it looks for:

Over-replicated blocks

These are blocks that exceed their target replication for the file they belong to. Normally, over-replication is not a problem, and HDFS will automatically delete excess replicas.

Under-replicated blocks

These are blocks that do not meet their target replication for the file they belong to. HDFS will automatically create new replicas of under-replicated blocks until they meet the

target replication. You can get information about the blocks being replicated (or waiting to be replicated) using hadoop dfsadmin -metasave.

Misreplicated blocks

These are blocks that do not satisfy the block replica placement policy. For example, for a replication level of three in a multirack cluster, if all three replicas of a block are on the same rack, then the block is misreplicated because the replicas should be spread across at least two racks for resilience. HDFS will automatically re-replicate misreplicated blocks so that they satisfy the rack placement policy.

Corrupt blocks

These are blocks whose replicas are all corrupt. Blocks with at least one noncorrupt replica are not reported as corrupt; the namenode will replicate the noncorrupt replica until the target replication is met.

Missing replicas

These are blocks with no replicas anywhere in the cluster. Corrupt or missing blocks are the biggest cause for concern, as it means data has been lost. By default, fsck leaves files with corrupt or missing blocks, but you can tell it to perform one of the following actions on them:

• Move the affected files to the /lost+found directory in HDFS, using the -move option. Files are broken into chains of contiguous blocks to aid any salvaging efforts you may attempt.

• Delete the affected files, using the -delete option. Files cannot be recovered after being deleted.

➢ Balancer

Over time, the distribution of blocks across datanodes can become unbalanced. An unbalanced cluster can affect locality for MapReduce, and it puts a greater strain on the highly utilized datanodes, so it's best avoided.

The balancer program is a Hadoop daemon that redistributes blocks by moving them from overutilized datanodes to underutilized datanodes, while adhering to the block replica placement policy that makes data loss unlikely by placing block replicas on different racks. It moves blocks until the cluster is deemed to be balanced, which means that the utilization of every datanode (ratio of used space on the node to total capacity of the node) differs from the utilization of the cluster (ratio of used space on the cluster to total capacity of the cluster) by no more than a given threshold percentage. You can start the balancer with:

% **start-balancer.sh**

The -threshold argument specifies the threshold percentage that defines what it means for the cluster to be balanced. The flag is optional, in which case the threshold is 10%.

At any one time, only one balancer may be running on the cluster. The balancer runs until the cluster is balanced; it cannot move any more blocks, or it loses contact with the namenode. It produces a logfile in the standard log directory, where it writes a line for every iteration of redistribution that it carries out. Here is the output from a short run on a small cluster:

Time Stamp Iteration# Bytes Already Moved Bytes Left To Move Bytes Being Moved

Mar 18, 2009 5:23:42 PM 0 0 KB 219.21 MB 150.29 MB
Mar 18, 2009 5:27:14 PM 1 195.24 MB 22.45 MB 150.29 MB
The cluster is balanced. Exiting...
Balancing took 6.072933333333333 minutes
The balancer is designed to run in the background without unduly taxing the cluster or interfering with other clients using the cluster. It limits the bandwidth that it uses to copy a block from one node to another. The default is a modest 1 MB/s, but this can be changed by setting the dfs.balance.bandwidthPerSec property in hdfs-site.xml, specified in bytes.

Monitoring

Monitoring is an important part of system administration. In this section, we look at the monitoring facilities in Hadoop and how they can hook into external monitoring systems.

The purpose of monitoring is to detect when the cluster is not providing the expected level of service. The master daemons are the most important to monitor: the namenode's (primary and secondary) and the jobtracker. Failure of datanodes and tasktrackers is to be expected, particularly on larger clusters, so you should provide extra capacity so that the cluster can tolerate having a small percentage of dead nodes at any time.

In addition to the facilities described next, some administrators run test jobs on a periodic basis as a test of the cluster's health.

Though it is not covered here, there is a lot of work going on to add more monitoring capabilities to Hadoop. For example, Chukwa, a data collection and monitoring system built on HDFS and MapReduce, excels at mining log data for finding large-scale trends.

Logging

All Hadoop daemons produce logfiles that can be very useful for finding out what is happening in the system. explains how to configure these files. Setting log levels When debugging a problem, it is very convenient to be able to change the log level temporarily for a particular component in the system.

Hadoop daemons have a web page for changing the log level for any log4j log name, which can be found at /logLevel in the daemon's web UI. By convention, log names in Hadoop correspond to the classname doing the logging, although there are exceptions to this rule, so you should consult the source code to find log names.

For example, to enable debug logging for the JobTracker class, we would visit the jobtracker's web UI at http://jobtracker-host:50030/logLevel and set the log name org.apache.hadoop.mapred.JobTracker to level DEBUG.

The same thing can be achieved from the command line as follows:

% **hadoop daemonlog -setlevel jobtracker-host:50030 **
org.apache.hadoop.mapred.JobTracker DEBUG

Log levels changed in this way are reset when the daemon restarts, which is usually what you want. However, to make a persistent change to a log level, simply change the log4j.properties file in the configuration directory. In this case, the line to add is: log4j.logger.org.apache.hadoop.mapred.JobTracker=DEBUG Getting stack traces Hadoop

daemons expose a web page (/stacks in the web UI) that produces a thread dump for all running threads in the daemon's JVM. For example, you can get a thread dump for a jobtracker from http://jobtracker-host:50030/stacks.

➢ Metrics

The HDFS and MapReduce daemons collect information about events and measurements that are collectively known as metrics. For example, datanodes collect the following metrics (and many more): the number of bytes written, the number of blocks replicated, and the number of read requests from clients (both local and remote).

Metrics belong to a context, and Hadoop currently uses "dfs", "mapred", "rpc", and "jvm" contexts. Hadoop daemons usually collect metrics under several contexts. For example, datanodes collect metrics for the "dfs", "rpc", and "jvm" contexts.

How Do Metrics Differ from Counters?

The main difference is their scope: metrics are collected by Hadoop daemons, whereas counters are collected for MapReduce tasks and aggregated for the whole job. They have different audiences, too: broadly speaking, metrics are for administrators, and counters are for MapReduce users.

The way they are collected and aggregated is also different. Counters are a MapReduce feature, and the MapReduce system ensures that counter values are propagated from the tasktrackers where they are produced, back to the jobtracker, and finally back to the client running the MapReduce job. (Counters are propagated via RPC heartbeats; The collection mechanism for metrics is decoupled from the component that receives the updates, and there are various pluggable outputs, including local files, Ganglia, and JMX. The daemon collecting the metrics performs aggregation on them before they are sent to the output.

A context defines the unit of publication; you can choose to publish the "dfs" context, but not the "jvm" context, for instance. Metrics are configured in the conf/hadoopmetrics. properties file, and by default, all contexts are configured so they do not publish their metrics.

FAILURE RECOVERY

Recovering from a failed NameNode

Failures happen, and Hadoop has been designed to be quite resilient. The NameNode, unfortunately, remains a weak point. HDFS is out of commission if the NameNode is down. A common design for setting up a backup NameNode server is by reusing the SNN. After all, the SNN has similar hardware specs as the NameNode, and Hadoop should've already been installed with the same directory configurations. If we do some additional work of maintaining the SNN to be a functional mirror image of the NameNode, we can quickly start this backup machine as a NameNode instance in the case of a NameNode failure. Some manual intervention and time are necessary to start the backup node as the new NameNode, but at least we wouldn't lose any data.

SNN- Unfortunately, this common design also contributes to the misperception of the Secondary NameNode as a backup node. You can set up the backup node in a totally different machine from the NameNode and SNN, but that machine would be idle almost all the time.

NameNode keeps all the filesystem's metadata, including the FsImage and EditLog files, under the dfs.name.dir directory. Note that the SNN server doesn't use that directory at all. It downloads the system's metadata into the fs.checkpoint.dir directory and proceeds to merge FsImage and EditLog there. As the dfs.name.dir directory on the SNN is unused, we can expose it to the NameNode via the Network File System (NFS). We instruct the NameNode to always write to this mounted directory in addition to writing to the NameNode's local metadata directory. HDFS supports this ability to write the metadata to multiple directories.

Cluster Specification

Hadoop is designed to run on commodity hardware. That means that you are not tied to expensive, proprietary offerings from a single vendor; rather, you can choose standardized, commonly available hardware from any of a large range of vendors to build your cluster.
"Commodity" does not mean "low-end." Low-end machines often have cheap components, which have higher failure rates than more expensive (but still commodityclass) machines. When you are operating tens, hundreds, or thousands of machines, cheap components turn out to be a false economy, as the higher failure rate incurs a greater maintenance cost. On the other hand, large database-class machines are not recommended either, since they don't score well on the price/performance curve. And even though you would need fewer of them to build a cluster of comparable performance than one built of mid-range commodity hardware, when one did fail, it would have a bigger impact on the cluster because a larger proportion of the cluster hardware would be unavailable.

Processor

Two quad-core 2-2.5 GHz CPUs

Memory

16-24 GB ECC RAM1

Storage

Four 1 TB SATA disks

Network

Gigabit Ethernet

Although the hardware specification for your cluster will assuredly be different, Hadoop is designed to use multiple cores and disks, so it will be able to take full advantage of more powerful hardware.

The bulk of Hadoop is written in Java and can therefore run on any platform with a JVM, although there are enough parts that harbor Unix assumptions (the control scripts, for example) to make it unwise to run on a non-Unix platform in production.

In fact, Windows operating systems are not supported production platforms (although they can be used with Cygwin as a development platform.

How large should your cluster be? There isn't an exact answer to this question, but the beauty of Hadoop is that you can start with a small cluster (say, 10 nodes) and grow it as your storage and computational needs grow. In many ways, a better question is this: how fast does my cluster need to grow? You can get a good feel for this by considering storage capacity.

For example, if your data grows by 1 TB a week and you have three-way HDFS replication, you need an additional 3 TB of raw storage per week. Allow some room for intermediate files and logfiles (around 30%, say), and this works out at about one (2010- vintage) machine per week, on average. In practice, you wouldn't buy a new machine each week and add it to the cluster. The value of doing a back-of-the-envelope calculation like this is that it gives you a feel for how big your cluster should be. In this example, a cluster that holds two years of data needs 100 machines.

For a small cluster (on the order of 10 nodes), it is usually acceptable to run the namenode and the jobtracker on a single master machine (as long as at least one copy of the namenode's metadata is stored on a remote filesystem). As the cluster and the number of files stored in HDFS grow, the namenode needs more memory, so the namenode and jobtracker should be moved onto separate machines. The secondary namenode can be run on the same machine as the namenode, but again

for reasons of memory usage (the secondary has the same memory requirements as the primary), it is best to run it on a separate piece of hardware, especially for larger clusters.

Machines running the namenodes should typically run on 64-bit hardware to avoid the 3 GB limit on Java heap size in 32-bit architectures.

NETWORK TOPOLOGY

A common Hadoop cluster architecture consists of a two-level network topology, as illustrated in below Fig I. Typically there are 30 to 40 servers per rack, with a
1 GB switch for the rack (only three are shown in the diagram) and an uplink to a core switch or router (which is normally 1 GB or better). The salient point is that the aggregate bandwidth between nodes on the same rack is much greater than that between nodes on different racks.

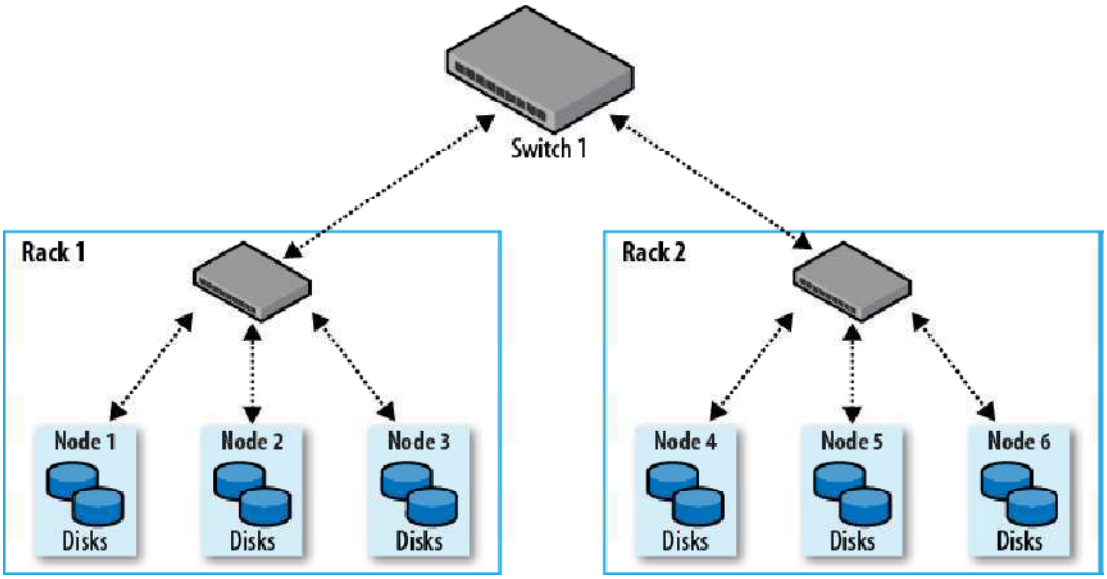


Fig I: Typical two-level network architecture for a Hadoop cluster

Rack awareness

To get maximum performance out of Hadoop, it is important to configure Hadoop so that it knows the topology of your network. If your cluster runs on a single rack, then there is nothing more to do, since this is the default. However, for multirack clusters, you need to map nodes to racks. By doing this, Hadoop will prefer within-rack transfers (where there is more bandwidth available) to off-rack transfers when placing MapReduce tasks on nodes. HDFS will be able to place replicas more intelligently to trade off performance and resilience.

Network locations such as nodes and racks are represented in a tree, which reflects the network "distance" between locations. The namenode uses the network location when determining where to place block replicas the MapReduce scheduler uses network location to determine where the closest replica is as input to a map task.

HADOOP BENCHMARKS

Hadoop comes with several benchmarks that you can run very easily with minimal setup cost. Benchmarks are packaged in the test JAR file, and you can get a list of them, with descriptions, by invoking the JAR file with no arguments:

% **hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar**

Most of the benchmarks show usage instructions when invoked with no arguments. For example:

% **hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO**

TestFDSIO.0.0.4

Usage: TestFDSIO -read | -write | -clean [-nrFiles N] [-fileSize MB] [-resFile

resultFileName] [-bufferSize Bytes]

Benchmarking HDFS with TestDFSIO

TestDFSIO tests the I/O performance of HDFS. It does this by using a MapReduce job as a convenient way to read or write files in parallel. Each file is read or written in a separate map task, and the output of the map is used for collecting statistics relating to the file just processed. The statistics are accumulated in the reduce, to produce a summary.

The following command writes 10 files of 1,000 MB each:

% **hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO -write -nrFiles 10**

-fileSize 1000

At the end of the run, the results are written to the console and also recorded in a local file (which is appended to, so you can rerun the benchmark and not lose old results):

% **cat TestDFSIO_results.log**

----- TestDFSIO ----- : write

Date & time: Sun Apr 12 07:14:09 EDT 2009

Number of files: 10

Total MBytes processed: 10000

Throughput mb/sec: 7.796340865378244

Average IO rate mb/sec: 7.8862199783325195

IO rate std deviation: 0.9101254683525547

Test exec time sec: 163.387

The files are written under the /benchmarks/TestDFSIO directory by default (this can be changed by setting the test.build.data system property), in a directory called io_data.

To run a read benchmark, use the -read argument. Note that these files must already exist (having been written by TestDFSIO -write):

% **hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO -read -nrFiles 10**

-fileSize 1000

Here are the results for a real run:

----- TestDFSIO ----- : read

Date & time: Sun Apr 12 07:24:28 EDT 2009

Number of files: 10

Total MBytes processed: 10000

Throughput mb/sec: 80.25553361904304

Average IO rate mb/sec: 98.6801528930664

IO rate std deviation: 36.63507598174921

Test exec time sec: 47.624

When you've finished benchmarking, you can delete all the generated files from HDFS using the -clean argument:

% **hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar TestDFSIO -clean**

Benchmarking MapReduce with Sort

Hadoop comes with a MapReduce program that does a partial sort of its input. It is very useful for benchmarking the whole MapReduce system, as the full input dataset is transferred through the shuffle. The three steps are: generate some random data, perform the sort, then validate the results.

First we generate some random data using RandomWriter. It runs a MapReduce job with 10 maps per node, and each map generates (approximately) 10 GB of random binary data, with key and values of various sizes. You can change these values if you like by setting the properties test.randomwriter.maps_per_host and test.randomwrite.bytes_per_map. There are also settings for the size ranges of the keys and values; see RandomWriter for details.

Here's how to invoke RandomWriter (found in the example JAR file, not the test one) to write its output to a directory called random-data:

% **hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar randomwriter random-data**

Next we can run the Sort program:

% **hadoop jar $HADOOP_INSTALL/hadoop-*-examples.jar sort random-data sorted-data**

The overall execution time of the sort is the metric we are interested in, but it's instructive to watch the job's progress via the web UI (http://jobtracker-host:50030/), where you can get a feel for how long each phase of the job takes.

As a final sanity check, we validate the data in sorted-data is, in fact, correctly sorted:

% **hadoop jar $HADOOP_INSTALL/hadoop-*-test.jar testmapredsort -sortInput random-data **

-sortOutput sorted-data

This command runs the SortValidator program, which performs a series of checks on the unsorted and sorted data to check whether the sort is accurate. It reports the outcome to the console at the end of its run:

HADOOP SCHEDULERS

Since the pluggable scheduler was implemented, several scheduler algorithms have been developed for it. The sections that follow explore the various algorithms available and when it makes sense to use them.

FIFO scheduler

The original scheduling algorithm that was integrated within the JobTracker was called *FIFO*. In FIFO scheduling, a JobTracker pulled jobs from a work queue, oldest job first. This schedule had no concept of the priority or size of the job, but the approach was simple to implement and efficient.

Capacity scheduler

The capacity scheduler shares some of the principles of the fair scheduler but has distinct differences, too. First, capacity scheduling was defined for large clusters, which may have multiple, independent consumers and target applications. For this reason, capacity scheduling provides greater control as well as the ability to provide a minimum capacity guarantee and share excess capacity among users. The capacity scheduler was developed by Yahoo!.

In capacity scheduling, instead of pools, several queues are created, each with a configurable number of map and reduce slots. Each queue is also assigned a guaranteed capacity (where the overall capacity of the cluster is the sum of each queue's capacity).

Queues are monitored; if a queue is not consuming its allocated capacity, this excess capacity can be temporarily allocated to other queues. Given that queues can represent a person or larger organization, any available capacity is redistributed for use by other users.

Another difference of fair scheduling is the ability to prioritize jobs within a queue. Generally, jobs with a higher priority have access to resources sooner than lower-priority jobs. The Hadoop road map includes a desire to support preemption (where a low-priority job could be temporarily swapped out to allow a higher-priority job to execute), but this functionality has not yet been implemented.

Another difference is the presence of strict access controls on queues (given that queues are tied to a person or organization). These access controls are defined on a per-queue basis. They restrict the ability to submit jobs to queues and the ability to view and modify jobs in queues.

You configure the capacity scheduler within multiple Hadoop configuration files. The queues are defined within hadoop-site.xml, and the queue configurations are set in capacity-scheduler.xml. You can configure ACLs within mapred-queue-acls.xml. Individual queue properties include capacity percentage (where the capacity of all queues in the cluster is less

than or equal to 100), the maximum capacity (limit for a queue's use of excess capacity), and whether the queue supports priorities. Most importantly, these queue properties can be manipulated at run time, allowing them to change and avoid disruptions in cluster use.

FAIR SCHEDULER

Introduction

Fair scheduling is a method of assigning resources to jobs such that all jobs get, on average, an equal share of resources over time. When there is a single job running, that job uses the entire cluster. When other jobs are submitted, tasks slots that free up are assigned to the new jobs, so that each job gets roughly the same amount of CPU time. Unlike the default Hadoop scheduler, which forms a queue of jobs, this lets short jobs finish in reasonable time while not starving long jobs. It is also an easy way to share a cluster between multiple of users. Fair sharing can also work with job priorities - the priorities are used as weights to determine the fraction of total compute time that each job gets.

The fair scheduler organizes jobs into pools, and divides resources fairly between these pools. By default, there is a separate pool for each user, so that each user gets an equal share of the cluster. It is also possible to set a job's pool based on the user's Unix group or any jobconf property. Within each pool, jobs can be scheduled using either fair sharing or first-in-first-out (FIFO) scheduling.

In addition to providing fair sharing, the Fair Scheduler allows assigning guaranteed minimum shares to pools, which is useful for ensuring that certain users, groups or production applications always get sufficient resources. When a pool contains jobs, it gets at least its minimum share, but when the pool does not need its full guaranteed share, the excess is split between other pools.

If a pool's minimum share is not met for some period of time, the scheduler optionally supports preemption of jobs in other pools. The pool will be allowed to kill tasks from other pools to make room to run. Preemption can be used to guarantee that "production" jobs are not starved while also allowing the Hadoop cluster to also be used for experimental and research jobs. In addition, a pool can also be allowed to preempt tasks if it is below half of its fair share for a configurable timeout (generally set larger than the minimum share preemption timeout). When choosing tasks to kill, the fair scheduler picks the most-recently-launched tasks from over-allocated jobs, to minimize wasted computation. Preemption does not cause the preempted jobs to fail, because Hadoop jobs tolerate losing tasks; it only makes them take longer to finish.

The Fair Scheduler can limit the number of concurrent running jobs per user and per pool. This can be useful when a user must submit hundreds of jobs at once, or for ensuring that
intermediate data does not fill up disk space on a cluster when too many concurrent jobs are running. Setting job limits causes jobs submitted beyond the limit to wait until some of the

user/pool's earlier jobs finish. Jobs to run from each user/pool are chosen in order of priority and then submit time.

Finally, the Fair Scheduler can limit the number of concurrent running tasks per pool. This can be useful when jobs have a dependency on an external service like a database or web service that could be overloaded if too many map or reduce tasks are run at once.

Installation

To run the fair scheduler in your Hadoop installation, you need to put it on the CLASSPATH. The easiest way is to copy the hadoop-fairscheduler-*.jar from HADOOP_HOME/build/contrib/fairscheduler to HADOOP_HOME/lib. Alternatively you can modify HADOOP_CLASSPATH to include this jar, in HADOOP_CONF_DIR/hadoop-env.sh

You will also need to set the following property in the Hadoop config file HADOOP_CONF_DIR/mapred-site.xml to have Hadoop use the fair scheduler:

```
<property>
  <name>mapred.jobtracker.taskScheduler</name>
  <value>org.apache.hadoop.mapred.FairScheduler</value>
</property>
```

Once you restart the cluster, you can check that the fair scheduler is running by going to http://<jobtracker URL>/scheduler on the JobTracker's web UI. A "job scheduler administration" page should be visible there. This page is described in the Administration section.

If you wish to compile the fair scheduler from source, run ant package in your HADOOP_HOME directory. This will build build/contrib/fair-scheduler/hadoop-fairscheduler-*.jar.

Configuration

The Fair Scheduler contains configuration in two places -- algorithm parameters are set in mapred-site.xml, while a separate XML file called the allocation file can be used to configure pools, minimum shares, running job limits and preemption timeouts. The allocation file is reloaded periodically at runtime, allowing you to change pool settings without restarting your Hadoop cluster.

For a minimal installation, to just get equal sharing between users, you will not need to set up an allocation file. If you do set up an allocation file, you will need to tell the scheduler where to find it by setting the mapred.fairscheduler.allocation.file parameter in mapred-site.xml as described below.

Scheduler Parameters in mapred-site.xml

The following parameters can be set in mapred-site.xml to affect the behavior of the fair scheduler:

Basic Parameters:

Name	Description
mapred.fairscheduler.allocation.file	Specifies an absolute path to an XML file which contains minimum shares for each pool, per-pool and per-user limits on number of running jobs, and preemption timeouts. If this property is not set, these features are not used. The allocation file format is described later.
mapred.fairscheduler.preemption	Boolean property for enabling preemption. Default: false.
mapred.fairscheduler.pool	Specify the pool that a job belongs in. If this is specified then mapred.fairscheduler.poolnameproperty is ignored.
mapred.fairscheduler.poolnameproperty	Specify which jobconf property is used to determine the pool that a job belongs in. String, default: user.name (i.e. one pool for each user). Another useful value is mapred.job.queue.name to use MapReduce's "queue" system for access control lists (see below). mapred.fairscheduler.poolnameproperty is used only for jobs in which mapred.fairscheduler.pool is not explicitly set.
mapred.fairscheduler.allow.undeclared.pools	Boolean property for enabling job submission to pools not declared in the allocation file. Default: true.

Advanced Parameters:

Name	Description
mapred.fairscheduler.sizebasedweight	Take into account job sizes in calculating their weights for fair sharing. By default, weights are
	only based on job priorities. Setting this flag to true will make them based on the size of the job (number of tasks needed) as well,though not linearly (the weight will be proportional to the log of the number of tasks needed). This lets larger jobs get larger fair shares while still providing enough of a share to small jobs to let them finish fast. Boolean value, default: false.

mapred.fairscheduler.preemption.only.log	This flag will cause the scheduler to run through The messages look as follows: `Should preempt 2 tasks for job_20090101337_0001: tasksDueToMinShare = 2, tasksDueToFairShare = 0`
mapred.fairscheduler.update.interval	Interval at which to update fair share 500.
mapred.fairscheduler.preemption.interval	Interval at which to check for tasks to preempt. t becomes less than the inter- milliseconds, default: 15000.
mapred.fairscheduler.weightadjuster	An extension point that lets you specify a class There is currently one example implementation - NewJobWeightBooster, which increases the
	weight of jobs for the first 5 minutes of their name, `org.apache.hadoop.mapred.NewJobWeightBooster` • Default is 3. • minutes.
mapred.fairscheduler.loadmanager	An extension point that lets you specify a class

	that determines how many maps and reduces can run on a given TaskTracker. This class should implement the LoadManager interface. By default the task caps in the Hadoop config file are used, but this option could be used to make the load based on available memory and CPU utilization for example.
mapred.fairscheduler.taskselector	An extension point that lets you specify a class that determines which task from within a job to launch on a given tracker. This can be used to change either the locality policy (e.g. keep some jobs within a particular rack) or the speculative execution algorithm (select when to launch speculative tasks). The default implementation uses Hadoop's default algorithms from JobInProgress.

Allocation File Format

The allocation file configures minimum shares, running job limits, weights and preemption timeouts for each pool. An example is provided in

HADOOP_HOME/conf/fair-scheduler.xml.template. The allocation file can contain the following types of elements:

- pool elements, which configure each pool. These may contain the following sub-elements:

    - minMaps and minReduces, to set the pool's minimum share of task slots.
    - maxMaps and maxReduces, to set the pool's maximum concurrent task slots.
    - schedulingMode, the pool's internal scheduling mode, which can be fair for fair sharing or fifo for first-in-first-out.
    - maxRunningJobs, to limit the number of jobs from the pool to run at once (defaults to infinite).

    - weight, to share the cluster non-proportionally with other pools (defaults to 1.0).
    - minSharePreemptionTimeout, the number of seconds the pool will wait before killing other pools' tasks if it is below its minimum share (defaults to infinite).

- user elements, which may contain a maxRunningJobs element to limit jobs. Note that by default, there is a pool for each user, so per-user limits are not necessary.

- poolMaxJobsDefault, which sets the default running job limit for any pools whose limit is not specified.
- userMaxJobsDefault, which sets the default running job limit for any users whose limit is not specified.
- defaultMinSharePreemptionTimeout, which sets the default minimum share preemption

timeout for any pools where it is not specified.

- fairSharePreemptionTimeout, which sets the preemption timeout used when jobs are below half their fair share.
- defaultPoolSchedulingMode, which sets the default scheduling mode (fair or fifo) for pools whose mode is not specified.

Pool and user elements only required if you are setting non-default values for the pool/user. That is, you do not need to declare all users and all pools in your config file before running the fair scheduler. If a user or pool is not listed in the config file, the default values for limits, preemption timeouts, etc will be used.

An example allocation file is given below :

```
<?xmlversion="1.0"?>
<allocations>

   <poolname="sample_pool">
      <minMaps>5</minMaps>
      <minReduces>5</minReduces>
      <maxMaps>25</maxMaps>
      <maxReduces>25</maxReduces>
      <weight>2.0</weight>
   </pool>

   <username="sample_user">
      <maxRunningJobs>6</maxRunningJobs>
   </user>
   <userMaxJobsDefault>3</userMaxJobsDefault>
</allocations>
```

This example creates a pool sample_pool with a guarantee of 5 map slots and 5 reduce slots. The pool also has a weight of 2.0, meaning it has a 2x higher share of the cluster than other pools (the default weight is 1). The pool has a cap of 25 map and 25 reduce slots, which means that once 25 tasks are running, no more will be scheduled even if the pool's fair share is higher. Finally, the example limits the number of running jobs per user to 3, except for sample_user, who can run 6 jobs concurrently. Any pool not defined in the allocation file will have no guaranteed capacity and a weight of 1.0. Also, any pool or user with no max running jobs set in the file will be allowed to run an unlimited number of jobs.
A more detailed example file, setting preemption timeouts as well, is available in HADOOP_HOME/conf/fair-scheduler.xml.template.

Access Control Lists (ACLs)

The fair scheduler can be used in tandem with the "queue" based access control system in MapReduce to restrict which pools each user can access. The fair scheduler to use one pool per queue by adding the following property in HADOOP_CONF_DIR/mapred-site.xml:

```
<property>
  <name>mapred.fairscheduler.poolnameproperty</name>
  <value>mapred.job.queue.name</value>
</property>
```

You can then set the minimum share, weight, and internal scheduling mode for each pool as described earlier. In addition, make sure that users submit jobs to the right queue by setting the mapred.job.queue.name property in their jobs.

Administration

The fair scheduler provides support for administration at runtime through two mechanisms:

1. It is possible to modify minimum shares, limits, weights, preemption timeouts and pool scheduling modes at runtime by editing the allocation file. The scheduler will reload this file 10-15 seconds after it sees that it was modified.

2. Current jobs, pools, and fair shares can be examined through the JobTracker's web interface, at http://<JobTracker URL>/scheduler. On this interface, it is also possible to modify jobs' priorities or move jobs from one pool to another and see the effects on the fair shares (this requires JavaScript).

The following fields can be seen for each job on the web interface:

- Submitted - Date and time job was submitted.
- JobID, User, Name - Job identifiers as on the standard web UI.
- Pool - Current pool of job. Select another value to move job to another pool.
- Priority - Current priority. Select another value to change the job's priority
- Maps/Reduces Finished: Number of tasks finished / total tasks.
- Maps/Reduces Running: Tasks currently running.
- Map/Reduce Fair Share: The average number of task slots that this job should

have at any given time according to fair sharing. The actual number of tasks will go up and down depending on how much compute time the job has had, but on average it will get its fair share amount.

In addition, it is possible to view an "advanced" version of the web UI by going to http://<JobTracker URL>/scheduler?advanced. This view shows two more columns:

- Maps/Reduce Weight: Weight of the job in the fair sharing calculations. This depends on priority and potentially also on job size and job age if the sizebasedweight and NewJobWeightBooster are enabled.

Metrics

The fair scheduler can export metrics using the Hadoop metrics interface. This can be enabled by adding an entry to hadoop-metrics.properties to enable the fairscheduler metrics context. For example, to simply retain the metrics in memory so they may be viewed in the /metrics servlet:

fairscheduler.class=org.apache.hadoop.metrics.spi.NoEmitMetricsContext

Metrics are generated for each pool and job, and contain the same information that is visible on the /scheduler web page.